The Chemical Approach to Typestate-Oriented Programming

The Chemical Approach to Typestate-Oriented Programming

Silvia Crafa
Università di Padova, Italy
crafa@math.unipd.it

Luca Padovani
Università di Torino, Italy
luca.padovani@di.unito.it

Abstract

We study a novel approach to typestate-oriented programming based on the chemical metaphor: state and operations on objects are molecules of messages and state transformations are chemical reactions. This approach allows us to investigate typestate in an inherently concurrent setting, whereby objects can be accessed and modified concurrently by several processes, each potentially changing only part of their state. We introduce a simple behavioral type theory to express in a uniform way both the private and the public interfaces of objects, to describe and enforce structured object protocols consisting of possibilities, prohibitions, and obligations, and to control object sharing.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Software/Program Verification*]: Class invariants; F.3.3 [*Studies of Program Constructs*]: Type structure

Keywords Typestate, concurrency, behavioral types, join calculus

1. Introduction

In an object-oriented program, the *interface* of an object describes the whole set of methods supported by the object throughout its entire lifetime. However, the usage of the object is more precisely explained in terms of its *protocol* [2], describing the sequences of method calls that are legal, possibly depending on the object's internal state. Typical examples of objects with structured protocols are files, iterators, and locks: a file can be read or written only after it has been opened; an iterator can be asked to access the next element of a collection only if such element has been verified to exist; a lock should be released if (and only if) it was previously acquired. Usually, such constraints on the legal sequences of method calls are only informally documented as *comments* along with method descriptions; in this form, however, they cannot be used by the compiler to detect protocol violations.

In [14], DeLine and Fähndrich have adapted the concept of *typestate* [47], originally introduced for imperative programs, to the object-oriented paradigm. Typestates are machine-understandable abstractions of an object's internal state that can be used (1) to identify the subset of fields and operations that are valid when the object is in some given state and (2) to specify the effect of such operations on the state itself. For example, on a file in state `CLOSED` the compiler would permit invocations of the `open` method and forbid invocations of the `read` method, whereas on a file in state `OPEN` it would only permit invocations of `read`, `write`, and `close` methods and forbid `open`. Furthermore, the type of `open` would be refined so as to specify that its invocation changes the state of the file from `CLOSED` to `OPEN`.

Typestate-oriented programming (TSOP for short) [1, 11, 21, 48] goes one step further and promotes typestates to a native feature of the programming language that encourages programmers to design objects around their protocol. Languages supporting TSOP provide explicit constructs for defining state-dependent object interfaces and implementations, for changing and possibly querying at runtime an object's typestate, and for annotating the signature of methods so as to describe their effect on the state of an object [1]. In order to track the points in the code where the state of an object changes, hence to detect – *at compile time* – potential violations of an object's protocol using typestate information, references to objects with structured protocols are required to be stored and shared in controlled ways. Not surprisingly, then, all languages supporting static typestate checking rely on more or less sophisticated forms of aliasing control [5] which may hinder the applicability of typestate to objects simultaneously accessed/modified by concurrent processes. Damiani et al. [11] have proposed an approach to conjugate typestate and concurrency in a Java-like language relying on some runtime support: users of an object can invoke any method at any time, even when the state of the object is uncertain; a method invocation is suspended until the object is in a state for which that method is legal; typestate information is used *within* methods, to make sure that only valid fields are accessed. This approach has both computational

and methodological costs: it requires all methods of a concurrent object to be synchronized, it limits parallelism by sequentializing all concurrent accesses to the same object, and it guarantees protocol compliance only within methods, where some form of aliasing control can be used.

The first contribution of this paper is a foundational study of TSOP in an *inherently concurrent* setting, whereby objects can be shared and accessed concurrently, and (portions of) their state can be changed while they are simultaneously used by several processes. We base our study on the Objective Join Calculus [16, 17] and we show that the idiomatic modeling of objects in the Objective Join Calculus has strong connections with the main TSOP features, including state-sensitive operations, explicit state change, and runtime state querying. Such connections draw heavily from the chemical metaphor that inspired the Objective Join Calculus: programs are modeled as chemical soups of molecules (*i.e.* multisets of messages sent to objects) that encode both the current state of the objects and the (pending) operations on them, while reaction rules correspond to object's methods definitions. In particular, chemical reactions explicitly specify both the valid combinations of state and operations as well as the changes performed by each operation on the state of an object. Incidentally, we observe that the Objective Join Calculus natively supports high-level concepts such as *compound* and *multidimensional* states [48]. This allows us to formally investigate the issues arising when states are partially/concurrently updated.

The second contribution of this paper is a theory of *behavioral types* for TSOP in the Objective Join Calculus and a corresponding *substructural type system*. We exploit the chemical metaphor once more to express in a unified and compositional way the combination of the encapsulated part of objects (state) and their public interface (operations) and to describe objects protocols in terms of the legit configurations of messages that the objects can/must handle. The key idea underpinning the type system is that distinct references to the same object may be given different types. This feature accounts for the fact that several processes may use the same object concurrently. For example, a lock could be shared by two processes P and Q and be acquired by one of them, say P. Then, the reference to the lock held by P would have a type stating that P must (eventually) release the object, whereas the reference to the lock held by Q would have a type stating that Q can (but need not) attempt to acquire the lock. The *overall* type of the lock would be the *combination* of these two types, defined in terms of a suitable behavioral connective. It should be remarked that the type of an object is defined once and for all by the programmer and does *not* depend on the number of references to the object. In fact, the mismatch between an object's unique type and the combination of the types of all of its references is explained in terms of a *behavioral subtyping relation*. Such relation serves other purposes as well: aside from realizing the obvious form of subtype polymorphism, it supports the characterization of safe/partial concurrent state transitions and it provides – at no additional cost – a key tool for deriving the protocol of objects with uncertain state. On objects without typestates, subtyping collapses to the traditional one.

```
1 def o = FREE | acquire(r) ▷ o.BUSY | r.reply(o)
2   or   BUSY | release    ▷ o.FREE
3 in o.FREE
4  | def c = reply(o') ▷ o'.release in o.acquire(c)
```

Listing 1. A lock in the Objective Join Calculus.

With these ingredients in place, we are able to provide a simple static analysis ensuring that well-typed, concurrent processes comply with the protocols of the objects they use. Compared to [11], our approach enables a fine-grained tuning of the kind of concurrency allowed on objects with structured protocols: non-aliased objects can fully benefit from *static typestate checking*, while shared/aliased objects can rely on *runtime synchronization* to resolve races and execute methods at the right time. The typing of objects regulates the tradeoff between these extremes and allow for a whole range of intermediate scenarios.

Structure of the Paper. We start with an informal overview of TSOP in the Objective Join Calculus (Section 2) before presenting its syntax and semantics (Section 3). We define syntax and semantics of types (Section 4), we describe the rules of the type system (Section 5), and comment on its safety properties (Section 6). In the latter part of the paper, we discuss the key aspects to implement the proposed framework (Section 7), we discuss related work (Section 8) as well as future research directions (Section 9). Proofs of the results can be found in the companion technical report [10].

2. The Chemistry of Typestates

The Chemical Metaphor. The Join Calculus [16, 17] originates from the Chemical Abstract Machine [4], a formal model of computations as sequences of chemical reactions transforming molecules. The Objective Join Calculus [19] is a mildly sugared version of the Join Calculus with object-oriented features: a program is made of a set of *objects* and a *chemical soup* of messages that can combine into complex molecules; each object consists of *reaction rules* corresponding to its methods; reaction rules are made of a *pattern* and a *body*: when a molecule in the chemical soup matches the pattern of a reaction, the molecule is consumed and the corresponding body produces other molecules.

Listing 1 shows the idiomatic implementation and use of a *lock* in the Objective Join Calculus. The definition on lines 1-2 creates a new object `o` with two reaction rules, separated by `or`. The symbol ▷ separates the pattern from the body of each rule, while | combines messages into complex molecules. The first reaction "fires" if a `FREE` message and an `acquire` message (with argument `r`) are sent to `o`: the

two messages are consumed and those on the right hand side of ▷ are produced. In this case, the argument `r` of `acquire` is a reference to another object representing the process that wants to acquire the lock. Hence the effect of triggering the first reaction is that a `BUSY` message is sent to `o` (in jargon, to "self") and a `reply` message is sent to `r` to notify the receiver that the lock has been successfully acquired. The second reaction specifies that the object can also consume a molecule consisting of a `BUSY` message and a `release` message. The reaction just sends a `FREE` message to `o`. The lock is initialized on line 3, by sending a `FREE` message to `o`.

The process on line 4 shows a typical use of the lock. Since communication in the Join Calculus is asynchronous, sequential composition is modeled by means of *continuation passing*: the process creates a continuation object `c` that reacts to the `reply` message sent by the lock; then, the process manifests its intention to acquire the lock by sending `acquire(c)` to `o`. When the reaction on line 1 fires, the `reply` triggers the reaction in `c` on line 4, causing the lock to be released. One aspect not explained in the above description is the passing of `o` in the `reply` message on line 1 which is bound to `o'` on line 4. Since on line 1 `o` corresponds to "self", sending `o` in the message `reply(o)` enables *method chaining*. In fact, with some appropriate syntactic sugar we could rewrite the process on line 4 just as

```
o.acquire.release
```

We will introduce a generalization of such syntactic sugar later on (see Example 3.3 and Listing 3). We will also see that method chaining is not just a trick for writing compact code, but is a key feature that our type system hinges on.

In the next section we will discuss a more complex use case (Example 3.4) where the lock defined in lines 1-2 is shared by two processes that compete for acquiring it. In that case, we will see that the complex molecules in the patterns of the lock's reaction rules are essential to make sure that the lock behaves correctly, namely that only one process can hold the lock at any time. In particular, if an `acquire(c')` message is available but there is no `FREE` message in the soup (because another process has previously acquired the lock thereby consuming `FREE`), the reaction in line 1 cannot fire and the process waiting for the `reply` message on `c'` is suspended until the lock is released.

State and Operations in the Join Calculus. Listing 1 provides a clear illustration of TSOP in the Join Calculus: a lock is either free or busy; it can only be acquired when it is free, and it can only be released when it is busy; acquisition makes the lock busy, and release makes it free again. The compound molecules in the patterns specify the valid combinations of state and operations, and the state is explicitly changed within the body of reactions.

These observations lead to a natural classification of messages in two categories: `FREE` and `BUSY` encode the *state* of the lock, while `acquire` and `release` represent its *operations* (we follow the convention that "state" messages are written in upper case and "operation" messages in lower case). Ideally, lock users should not even be aware of the existence of `FREE` and `BUSY`, if only to prevent accidental or malicious violations of the lock protocol. Our type system will enforce an encapsulation mechanism to prevent users from sending state messages (Section 5).

Messages in the chemical soup encode the current state of the object and the (pending) operations on it: for instance, the presence of a message `o.FREE` in the soup encodes the fact that the object `o` *is in state* `FREE`; the presence of a message `o.acquire` in the soup encodes the fact that *there is a pending invocation* to the `acquire` method of the object `o`. Representing state using (molecules of) messages makes it simple to model so-called *and-states* [48], which we will see at work in Example 5.8. On the contrary, `FREE` and `BUSY` are examples *or-states* which mutually exclude each other. The typing of the lock object will guarantee that there is always exactly one message among `FREE` and `BUSY`, *i.e.* that the state of the lock is always uniquely determined.

Behavioral Types for the Join Calculus. Since in the Join Calculus there is no sharp distinction between (private) messages that encode the object's state and (public) messages that represent the object's operations, we can devise a type language to describe the legit configurations of messages, both private and public, that objects can/must handle. In fact, we can use types to specify (and enforce) the object protocol. Object types are built from message types $\mathtt{m}(\tilde{t})$ using three behavioral connectives, the *product* $\otimes$, the *choice* $\oplus$, and the *exponential* $*$. An object of type $\mathtt{m}(\tilde{t})$ *must* be used for sending an `m`-tagged message with a (possibly empty) tuple of arguments of type $\tilde{t}$; an object of type $t \otimes s$ *must* be used **both** as specified by t **and** as specified by s; an object of type $t \oplus s$ *must* be used **either** as specified by t **or** as specified by s; an object of type $*t$ *can* be used any number of times (even zero), each time as specified by t.

As an example, let us illustrate the type of the lock object. It is useful to keep in mind the intuition that the type of the lock should describe the whole set of legit configurations of messages targeted to the lock. In this respect, we recall that:

- there *must* be exactly one message among `FREE` and `BUSY` that encodes the state of the lock;
- there *can* be an arbitrary number of `acquire` messages regardless of the state of the lock (the lock is useful only if it is shared among several processes);
- there *must* be one `release` message if the lock is `BUSY` (this is an eventual obligation).

We express all these constraints with the type

$$t_{lock} \stackrel{\text{def}}{=} *\mathtt{acquire(reply(release))} \otimes \big(\mathtt{FREE} \oplus (\mathtt{BUSY} \otimes \mathtt{release})\big)$$

It is no coincidence that the only occurrence of $*$ is used in front of the only message (`acquire`) for which there

P, Q	::=		**Process**
		`null`	(null process)
	|	$u\,.\,M$	(message sending)
	|	$P \mid Q$	(process composition)
	|	`def` a = C `in` P	(object definition)
M, N	::=		**Molecule / Pattern**
		`m`$(\tilde{u})$	(message)
	|	$M \mid N$	(molecule composition)
C, D	::=		**Class**
		$J \triangleright P$	(reaction rule)
	|	C `or` D	(class composition)

Table 1. Syntax of the Objective Join Calculus.

are no obligations: the lock *can* but need not be acquired. However, if the lock is acquired, then it *must* be released; whence the lack of $*$ in front of `release`. There is no $*$ in front of `FREE` and `BUSY` either, meaning that there is an obligation to produce these messages too, but since `FREE` and `BUSY` occur in different branches of a $\oplus$ type, only one of them must be produced. In addition to possibilities and obligations, t_{lock} expresses prohibitions: all message configurations containing multiple `FREE` or `BUSY` messages or both `FREE` and `release` messages are prohibited by the type. Our type system will guarantee that any lock object is always in a configuration that is legal according to t_{lock}. This implies, for example, that a well-typed process never attempts to release a lock that is in state `FREE`.

There is one last thing to discuss before we end this informal overview, that is the type of the argument of `acquire`, named `r` in Listing 1. If we look at the code, we see that `r` is the reference to an object to which the lock sends a `reply(o)` message. Not surprisingly then, the argument of `acquire` has type `reply(release)` in t_{lock}. This means that the reference `o'` in Listing 1 has type `release`, which is consistent with the way it is used on line 4. In other words, we use method chaining to express the change in the (public) type of an object as methods are invoked. Both `o` and `o'` refer to the same lock object, but they have different interfaces: the former can be used for acquiring the lock; the latter must be used (once) for releasing it.

3. The Objective Join Calculus

The syntax of the Objective Join Calculus is defined in Table 1. We assume countable sets of *object names* a, b, c, ... and of *variables* x, y, ... We let u, v, ... denote *names*, which are either object names or variables, and use `m`, ... to range over *message tags*. We write $\tilde{u}$ for a (possibly empty) tuple $u_1, \ldots, u_n$ of names; we will use this notation extensively for denoting tuples of various entities. In a few occasions, we will also use $\tilde{u}$ as the set of names in $\tilde{u}$.

The syntax of the calculus comprises the syntactic categories of *processes*, *molecules*, and *classes*. Molecules are assemblies of messages and each message `m`$(\tilde{u})$ is made of a tag `m` and a tuple $\tilde{u}$ of arguments; we will abbreviate `m()` with `m`; *join patterns* (or simply patterns) J, ... are molecules whose arguments are all variables and that satisfy two *linearity conditions*: variables and message tags occurring in the same pattern are pairwise distinct. These conditions are typical of most presentations of the Join Calculus and are motivated by efficiency reasons: variable linearity avoids the need for equality tests when matching molecules; tag linearity allows the implementation to represent the state of each message queue with just one bit, according to whether there is no message or at least one message with a given tag. In our case, tag linearity is in fact necessary for the soundness of the type system (see Remark 5.4).

The process `null` is inert and does nothing. The process $u\,.\,M$ sends the messages in the molecule M to u. The process $P \mid Q$ is the parallel composition of P and Q. Finally, `def` a = C `in` P creates a new instance a of the class C. The name a is bound both in C (where it plays the role of "self") and in P. A class is a disjunction of *reaction rules*, which we will often represent as a set $\{J_i \triangleright P_i\}_{i \in I}$. Each rule consists of a *pattern* J_i and a *body* P_i. The variables in J_i are bound in P_i. An instance of P_i is spawned each time a molecule matching J_i is sent to an object that is instance of the class.

We omit the formal definition of free and bound names, which can be found in [19]. We write $\mathsf{fn}(P)$ for the set of free names in P and we identify processes up to renaming of bound names. In this paper we use an additional constraint, which is not restrictive and simplifies the type system: we require classes to have no free names other than "self".

We now turn to the operational semantics of the calculus, which describes the evolution of a *solution* $\mathscr{D} \Vdash \mathscr{P}$ made of a set $\mathscr{D} = \{a_i = C_i\}_{i \in I}$ of object definitions and a multiset $\mathscr{P}$ of parallel processes. Intuitively, $\mathscr{P}$ is a "soup" of processes and molecules that is subject to changes in the temperature (expressed by a relation $\rightleftharpoons$) and reactions (expressed by a relation $\rightarrow$). Heating $\rightharpoonup$ breaks things apart, while cooling $\rightharpoondown$ recombines them together, in possibly different configurations. Heating and cooling are reversible transformations of the soup, defined by the first four rules in Table 2: rule [NULL] states that `null` processes may evaporate or condense; rule [DEF] moves objects definitions to/from the $\mathscr{D}$ component of solutions, having care not to capture free names (disposing of a countable set of object names, we can always silently perform suitable alpha-renamings to avoid captures); rule [COMP-1] breaks and recombines processes and rule [COMP-2] does the same with molecules. To avoid unnecessary clutter, following [19], in all rules except [DEF] we omit unaffected definitions and processes. In rule [DEF], it is important to mention the whole set of definitions $\mathscr{D}$ to

[NULL]		$\Vdash$ `null`	$\rightleftharpoons$		$\Vdash$	
[DEF]	$\mathscr{D}$	$\Vdash$ $\mathscr{P}$, `def` $a = C$ `in` P	$\rightleftharpoons$	$\mathscr{D}, a = C$	$\Vdash$ $\mathscr{P}, P$	$a \notin \mathsf{fn}(\mathscr{P})$
[COMP-1]		$\Vdash$ $P \mid Q$	$\rightleftharpoons$		$\Vdash$ P, Q	
[COMP-2]		$\Vdash$ $a.(M \mid N)$	$\rightleftharpoons$		$\Vdash$ $a.M, a.N$	
[RED]	$a = \{J_i \triangleright P_i\}_{i \in I}$	$\Vdash$ $a.\sigma J_k$	$\rightarrow$	$a = \{J_i \triangleright P_i\}_{i \in I}$	$\Vdash$ σP_k	$k \in I$

Table 2. Reduction semantics of the Objective Join Calculus.

make sure that the name a of the object being defined is fresh. Rule [RED] defines reactions as non-reversible transformations of the soup. A reaction may happen whenever the soup contains a molecule targeted to some object a such that the shape of the molecule matches the pattern of one of the rules in the class of a, up to some substitution σ mapping variables to object names (recall that $\{J_i \triangleright P_i\}_{i \in I}$ stands for an `or`-composition of reaction rules, as by Table 1). In this case, the molecule is consumed by the reaction and replaced by the body of the rule, with the substitution σ applied.

Remark 3.1. The operational semantics presents three forms of non-determinism, due to the heating/cooling of molecules in the soup, the interleaving of reactions pertaining different objects, and the choice of reactions pertaining each single object. The first form of non-determinism is only relevant in the formal model. In practice, it is resolved since the Objective Join Calculus enjoys *locality*: each reaction involves messages targeted to the *same* object, therefore all messages sent to an object a travel to and react at the *unique* location of a. The second form of non-determinism accounts for the concurrent setting that we are modeling: when the soup contains molecules that can trigger reactions pertaining different objects, these reactions may fire in any order or even simultaneously, depending on the system architecture. The last form of non-determinism arises when there are enough molecules in the soup to trigger different reactions pertaining the same object. This is usually resolved by the compiler, which translates join patterns into code that processes the messages targeted at one given object according to a deterministic scheduler. In this case, the fact that the formal operational semantics is underspecified (*i.e.* non-deterministic) accounts for all possible implementations of join patterns. ■

In the rest of the section we illustrate the calculus by means of examples. For better clarity, we augment the calculus with conditionals and a few native data types, which can be either encoded or added without difficulties.

Example 3.2 (iterator)**.** Listing 2 shows a possible modeling of an array iterator class in the Objective Join Calculus. Like in object-based languages, the class is modeled as an object `ArrayIterator` providing just one factory method, `new` (line 1), whose arguments are an array `a` and a continuation object `r` to which the fresh instance of the iterator is sent. The iterator itself is an object `o` that can be in one of three states, `INIT`, `SOME`, or `NONE`. States `INIT` and `SOME` have arguments `a` (the array being iterated) and `n` (the index of the current element in the array). `INIT` is a transient state used for initializing the iterator (lines 3–4): the iterator spontaneously moves into either state `SOME` or state `NONE`, depending on whether `n` is smaller than the length `#a` of the array or not. When in state `SOME`, the iterator provides a `next` operation (lines 5–6) for reading the current element `a[n]` of the array and moving onto the next one. Since `n` might be the index of the last element of the array, the iterator transits to state `INIT`, which appropriately re-initializes the iterator. The iterator also provides a `peek` operation that can be used for querying the state of the iterator (lines 7–8). The operation does not change the state of the iterator and sends a message on the continuation `r` with either tag `some` or tag `none`, depending on the internal state of the iterator. ■

```
1  def ArrayIterator = new(a,r) ▷
2    def o =
3       INIT(a,n) ▷
4         if n < #a then o.SOME(a,n) else o.NONE
5    or SOME(a,n) | next(r) ▷
6         o.INIT(a,n+1) | r.reply(a[n],o)
7    or SOME(a,n) | peek(r) ▷ o.SOME(a,n) | r.some(o)
8    or NONE      | peek(r) ▷ o.NONE      | r.none(o)
9    in o.INIT(a,0) | r.reply(o)
10 in ...
```

Listing 2. An array iterator.

Example 3.3 (sequential composition)**.** In this example we see how to encode a sequential composition construct

$$\texttt{let}\ \tilde{u} = u.\texttt{m}(\tilde{v})\ \texttt{in}\ P$$

in the Objective Join Calculus. Intuitively, this construct invokes method `m` on object u with arguments $\tilde{v}$, waits for the results $\tilde{u}$ of the invocation, and continues as P. We let

$$\begin{array}{l}\texttt{let}\ \tilde{u} = u.\texttt{m}(\tilde{v})\ \texttt{in}\ P \stackrel{\text{def}}{=} \\ \quad \texttt{def}\ c = \texttt{WAIT}(\tilde{x}) \mid \texttt{reply}(\tilde{u}) \triangleright P\{\tilde{x}/\tilde{w}\} \\ \quad\ \texttt{in}\ c.\texttt{WAIT}(\tilde{w}) \mid u.\texttt{m}(\tilde{v}, c)\end{array}$$

where c and $\tilde{x}$ are fresh, $\tilde{w} = \mathsf{fn}(P) \backslash \tilde{u}$, and $P\{\tilde{x}/\tilde{w}\}$ denotes P where $\tilde{u}$ have been replaced by $\tilde{x}$. The twist in this encoding is that all the free names of P except $\tilde{u}$ are temporarily spilled into a message `WAIT` and then recovered when the callee sends the `reply` message on c. Normally, such spilling is not necessary in the encoding with continuation passing. We do it here to comply with our working assumption that classes have no free names other than "self".

```
1 def Lock = new(r) ▷
2  def o = FREE | acquire(r) ▷ o.BUSY | r.reply(o)
3     or   BUSY | release    ▷ o.FREE
4  in o.FREE | r.reply(o)
5 in let lock = Lock.new       (* lock : t_ACQUIRE *)
6 in let lock = lock.acquire   (* lock : t_RELEASE *)
7 in lock.release
```

Listing 3. Lock class definition.

```
1 def Philosopher = new(fork) ▷
2   def o = THINK | FORK(fork) ▷
3             o.FORK(fork) |
4             let f = fork.acquire in o.EAT(f)
5      or   EAT(f) ▷ o.THINK | f.release
6   in o.THINK | o.FORK(fork)
7 in let fork = Lock.new  (* fork : t_ACQUIRE *)
8 in Philosopher.new(fork) | Philosopher.new(fork)
```

Listing 4. Two dining philosophers.

Using this construct we rephrase the code of Listing 1 into that of Listing 3, which also encapsulates the lock definition into the `Lock` class. The re-binding of the `lock` name on lines 5 and 6 is typical of languages with explicit continuations [22]. An actual language would provide either adequate syntactic sugar or a native synchronous method call [17]. The types in comments will be described in Section 4. ■

Example 3.4 (dining philosophers)**.** We now discuss an example where the same lock is shared by two concurrent processes. Listing 4 models two philosophers that compete for the same fork when hungry. The fork is created on line 7 and shared by two instances of the `Philosopher` class (line 8). Each philosopher alternates between states `THINK` and `EAT`. In addition, the `FORK` message holds a reference to the shared fork and is meant to be an invariant part of each philosopher's state. Transitions occur non-deterministically: while in state `THINK`, the reaction on line 2 may fire; at that point, the philosopher restores the `FORK` message (line 3) and attempts to acquire the fork; when the fork is acquired, the philosopher transits into state `EAT` (line 4). While in state `EAT`, the philosopher holds a reference `f` to the acquired fork; when the reaction on line 5 fires, the fork is released and the philosopher goes back to state `THINK`. Note that this reaction consumes only part of the philosopher's state, which also comprises the `FORK` message. ■

4. Syntax and Semantics of Types

In this section we define a type language to describe object protocols in terms of the *valid configurations* of messages they accept. *Types* $t, s, \ldots$ are the regular trees [9] coinductively generated by the productions below:

$$t, s \;::=\; \mathbb{0} \mid \mathbb{1} \mid \mathtt{m}(\tilde{t}) \mid t \oplus s \mid t \otimes s \mid *t$$

The type $\mathtt{m}(\tilde{t})$ denotes an object that *must* be used for sending a message with tag $\mathtt{m}$ and arguments of type $\tilde{t}$; when $\tilde{t}$ is the empty tuple, we omit the parentheses altogether. Compound types are built using the behavioral connectives $\oplus$, $\otimes$, and $*$: an object of type $t \oplus s$ *must* be used either according to t or according to s; an object of type $t \otimes s$ *must* be used both according to t and also according to s; an object of type $*t$ *can* be used any number of times, each time according to t. Finally, we introduce the constants $\mathbb{0}$ and $\mathbb{1}$, which respectively represent the empty sum and the empty product. Intuitively, $\mathbb{0}$ is the type of all objects and $\mathbb{1}$ is the type of all objects without obligations. Occasionally we will also use basic types such as `int` and `real`.

Some examples: an object of type $\mathtt{m}(\mathtt{int})$ must be used for sending an $\mathtt{m}$ message with one argument of type `int`; an object of type $\mathtt{m}(\mathtt{int}) \oplus \mathbb{1}$ can be used for sending an $\mathtt{m}$ message, or it can be left alone; an object of type $\mathtt{m} \oplus \mathtt{m}'$ must be used for sending either an $\mathtt{m}$ message or an $\mathtt{m}'$ message, while an object of type $\mathtt{m} \otimes \mathtt{m}'$ must be used for sending both an $\mathtt{m}$ message and an $\mathtt{m}'$ message; finally, an object of type $*(\mathtt{m} \oplus \mathtt{m}')$ can be used for sending any number of $\mathtt{m}$ and $\mathtt{m}'$ messages. There is no legal way to use an object of type $\mathbb{0}$.

We do not devise an explicit syntax for recursive types. We work instead with (possibly infinite) regular trees directly. Recall that regular trees can always be finitely represented either as systems of equations or using the well-known μ notation; [9] is the standard reference for the theory of regular trees. For example, the type satisfying the equation $t = \mathbb{1} \oplus \mathtt{m}(t)$ denotes an object that can be used for sending an $\mathtt{m}$-tagged message with an argument which is itself an object with type t. We require every infinite branch of a type to go through infinitely many message type constructors. This condition (a strengthened contractiveness) excludes meaningless terms such as $t = t \oplus t$ or $t = *t$ and provides us with an induction principle on the structure of types that we will use in Definition 4.1 below.

We reserve some notation for useful families of types: we use $\mathtt{M}$ to range over *message types* $\mathtt{m}(\tilde{t})$ and T, S to range over *molecule types*, namely types of the form $\bigotimes_{i \in I} \mathtt{M}_i$; we identify molecule types modulo associativity and commutativity of $\otimes$ and product with $\mathbb{1}$; if $T = \bigotimes_{i \in I} \mathtt{m}_i(\tilde{t}_i)$, we write $\overline{T}$ for its *signature*, namely the multiset $\{\mathtt{m}_i\}_{i \in I}$.

The following definition formalizes the idea that types describe the valid configurations of messages that can be sent to objects. Whenever X and Y are sets of molecule types, we let $\mathsf{XY} \stackrel{\text{def}}{=} \{T \otimes S \mid T \in \mathsf{X} \wedge S \in \mathsf{Y}\}$ and we write X^n for the n-th power of X for $n \in \mathbb{N}$, where $\mathsf{X}^0 = \{\mathbb{1}\}$.

Definition 4.1 (valid configuration)**.** The *interpretation* of a type t, denoted by $[\![t]\!]$, is the set of molecule types inductively defined by the following equations:

$$[\![\mathbb{0}]\!] \stackrel{\text{def}}{=} \emptyset \qquad [\![t \oplus s]\!] \stackrel{\text{def}}{=} [\![t]\!] \cup [\![s]\!] \qquad [\![\mathtt{M}]\!] \stackrel{\text{def}}{=} \{\mathtt{M}\}$$
$$[\![\mathbb{1}]\!] \stackrel{\text{def}}{=} \{\mathbb{1}\} \qquad [\![t \otimes s]\!] \stackrel{\text{def}}{=} [\![t]\!][\![s]\!] \qquad [\![*t]\!] \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} [\![t]\!]^n$$

We say that T is a *valid configuration* for t if $T \in [\![t]\!]$.

For instance, $[\![\mathtt{m} \oplus \mathtt{m}']\!] = \{\mathtt{m}, \mathtt{m}'\}$ and $[\![\mathtt{m} \otimes \mathtt{m}']\!] = \{\mathtt{m} \otimes \mathtt{m}'\}$. Indeed, in the first case one can choose to send *either* $\mathtt{m}$ *or* $\mathtt{m}'$, whereas in the second case one must send *both*. Note that $\mathbb{0}$ has no valid configurations and that type $*t$ has, in general, infinitely many valid configurations. For instance, $[\![*\mathtt{m}]\!] = \{\mathbb{1}, \mathtt{m}, \mathtt{m} \otimes \mathtt{m}, \mathtt{m} \otimes \mathtt{m} \otimes \mathtt{m}, \dots\}$.

We have collected all the ingredients for defining the subtyping relation. Since types are possibly infinite terms, we must resort to a coinductive definition:

Definition 4.2 (subtyping)**.** We write $\leqslant$ for the largest relation between types such that $t \leqslant s$ and $\bigotimes_{i \in I} \mathtt{m}_i(\tilde{s}_i) \in [\![s]\!]$ imply $\bigotimes_{i \in I} \mathtt{m}_i(\tilde{t}_i) \in [\![t]\!]$ and $\tilde{s}_i \leqslant \tilde{t}_i$ for every $i \in I$. If $t \leqslant s$ holds, then we say that t is a *subtype* of s and s a *supertype* of t. We write $t \simeq s$ if $t \leqslant s$ and $s \leqslant t$.

To understand subtyping, it helps keeping in mind the usual safe substitution principle: when $t \leqslant s$, it is safe to use an object of type t where an object of type s is expected. In our setting, "using an object of type s" means sending to the object a message configuration that is valid for s. Definition 4.2 requires each valid configuration for s to also be a valid configuration for t, modulo contravariant subtyping of argument types. More specifically, whenever $S \in [\![s]\!]$, there exists some $T \in [\![t]\!]$ with the same signature as S such that the arguments of corresponding messages in T and S are related contravariantly. For instance, if $s = \mathtt{m}(\mathtt{int})$, then using an object of type s means sending to the object one message of the form $\mathtt{m}(n)$, where n is an integer number. Then, assuming $\mathtt{int} \leqslant \mathtt{real}$, it is safe to replace such object with another one of type $t = \mathtt{m}(\mathtt{real})$: the message $\mathtt{m}(n)$ sent to the former object will be understood without problems also by the latter object, as any integer number is also a real number. Therefore, $\mathtt{m}(\mathtt{real}) \leqslant \mathtt{m}(\mathtt{int})$.

We wish to reassure the reader bewildered by Definition 4.2 that $\leqslant$ shares many traits with conventional subtyping relations for object-oriented languages (see Example 4.5). Let us discuss a few notable relations in more detail.

- We have that $\mathtt{m} \oplus \mathtt{m}' \leqslant \mathtt{m}$. The user of an object of type $\mathtt{m}$ must send $\mathtt{m}$ to it. This is also a particular valid use (although not the only valid use) of an object of type $\mathtt{m} \oplus \mathtt{m}'$, which requires its users to send either $\mathtt{m}$ or $\mathtt{m}'$.
- We have $\mathtt{m} \otimes \mathtt{m}' \not\leqslant \mathtt{m}$ and $\mathtt{m} \not\leqslant \mathtt{m} \otimes \mathtt{m}'$. The user of an object of type $\mathtt{m} \otimes \mathtt{m}'$ must send both $\mathtt{m}$ and $\mathtt{m}'$, hence sending only $\mathtt{m}$ is an illegal way of using it. Vice versa, the user of an object of type $\mathtt{m}$ must send only $\mathtt{m}$, hence sending also $\mathtt{m}'$ is an illegal way of using it.
- We have that $\mathtt{m}(t) \leqslant \mathtt{m}(t \oplus s)$ and $\mathtt{m}(s) \leqslant \mathtt{m}(t \oplus s)$, namely $\mathtt{m}(t \oplus s)$ is an upper bound of both $\mathtt{m}(t)$ and $\mathtt{m}(s)$ (in fact, it is the *least* upper bound of these two types). The user of an object of type $\mathtt{m}(t \oplus s)$ must send $\mathtt{m}$ with an argument that *can* be used according to either t or s, hence this is also a valid use for an object of type $\mathtt{m}(t)$ or an object of type $\mathtt{m}(s)$. We will see a key instance of these relations in Example 4.7.

The interested reader can verify a number of additional useful properties: that $\mathbb{0}$ and $\mathbb{1}$ are indeed the units of $\oplus$ and $\otimes$; that $\mathbb{0}$ is absorbing for $\otimes$; that $\oplus$ distributes over $\otimes$. We capture all these properties by the following proposition.

Proposition 4.3. *The following properties hold:*

1. *$\leqslant$ is a pre-order and a pre-congruence;*
2. *the language of types taken modulo the $\simeq$ equivalence is a commutative Kleene algebra [8].*

We give a useful taxonomy of types: *linear* types denote objects that *must* be used; *non-linear* types denote objects without obligations; *usable* types denote objects that *can* be used, in the sense that there is a valid way of using them.

Definition 4.4 (type classification)**.** We say that t is *non linear*, notation $\mathsf{nl}(t)$, if $t \leqslant \mathbb{1}$; that t is *linear*, notation $\mathsf{lin}(t)$, if $t \not\leqslant \mathbb{1}$; that t is *usable*, notation $\mathsf{usable}(t)$, if $t \not\simeq \mathbb{0}$.

If $t \leqslant \mathbb{1}$, then $\mathbb{1} \in [\![t]\!]$ namely it is allowed not to send any message to an object of type t. If $t \simeq \mathbb{0}$, then t is linear but not usable, hence it denotes *absurd* objects that must be used, but at the same time such that there is no valid way of using them.

Example 4.5 (standard class type)**.** The class of a conventional object-oriented language can be described as the type $\bigotimes_{i \in I} *\mathtt{m}_i(\tilde{t}_i)$, saying that the objects of this class can be used for unlimited invocations of all of the available methods, in whatever order. Our subtyping relation is consistent with that typically adopted in such languages, since $\bigotimes_{i \in I} *\mathtt{m}_i(\tilde{t}_i) \leqslant \bigotimes_{j \in J} *\mathtt{m}_j(\tilde{s}_j)$ if and only if $I \supseteq J$ and $\tilde{t}_j \geqslant \tilde{s}_j$ for all $j \in J$ (the subclass has more methods, with arguments of larger type). ■

Example 4.6 (lock interfaces)**.** We illustrate the typing of the lock object used in Listings 1 and 3. Observe that the type t_{lock}, discussed in Section 2, describes both states and operations and is a valid type for the lock object as a whole. Correspondingly, t_{lock} can be correctly assigned to the binding occurrence of $\mathtt{o}$ on line 2 in Listing 3 according to the type system we will define in Section 5. Lock users are solely concerned with the public interfaces of the lock, which only refer to the `acquire` and `release` methods. We define:

$$t_{\mathtt{ACQUIRE}} \stackrel{\text{def}}{=} *\mathtt{acquire}(\mathtt{reply}(t_{\mathtt{RELEASE}}))$$
$$t_{\mathtt{RELEASE}} \stackrel{\text{def}}{=} \mathtt{release}$$

respectively for the interface of unacquired and acquired locks. Observe that $t_{\mathtt{ACQUIRE}}$ is non linear, indicating no obligations on unacquired locks (they can be used any number of times) whereas $t_{\mathtt{RELEASE}}$ is linear, indicating that acquired locks must be released (eventually). These interfaces can be "derived" (quite literally) by removing the state types from t_{lock}; we will make this relation precise in Section 5.

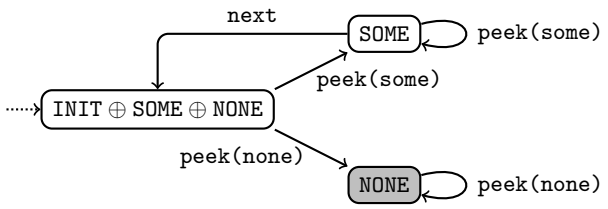

Figure 1. Transition diagram of the iterator.

The fact that (unacquired) locks can be shared without constraints is a consequence of the relation

$$t_{\texttt{ACQUIRE}} \simeq t_{\texttt{ACQUIRE}} \otimes t_{\texttt{ACQUIRE}}$$

stating a well-known property of the exponential/Kleene star. This property is precisely the one needed for typing the code in Listing 4, where one `fork` of type $t_{\texttt{ACQUIRE}}$ is created (line 7) and then shared by two philosophers (line 8). Thanks to this property, the type of a lock is independent of the number of processes trying to acquire it. Note that different references to the same lock, like the references to the fork shared by the two philosophers in Listing 4, may have different types corresponding to the different public interfaces exposed by the references. For instance, the reference `f` held by an eating philosopher has type $t_{\texttt{RELEASE}}$, hence it prescribes a release, while the reference `fork` held by a thinking philosopher has type $t_{\texttt{ACQUIRE}}$, hence it allows an acquisition. ■

Example 4.7 (iterator interfaces)**.** Let us consider the array iterator defined in Listing 2. We postpone the description of the whole type of the iterator object until Section 5, and we discuss here just the public interfaces exposed by the object in the different states, with the help of the transition diagram in Figure 1. When in state `NONE`, the iterator has reached the end of the array and there is only one method available, `peek`, which replies with a `none` message containing the iterator unchanged. Therefore, the public interface of the iterator in state `NONE` is the type satisfying the equation

$$t_{\texttt{NONE}} = \texttt{peek(none}(t_{\texttt{NONE}})) \oplus \mathbb{1}$$

The $\mathbb{1}$ term makes $t_{\texttt{NONE}}$ non linear, allowing the disposal of the iterator when in state `NONE`. Without it, linearity would force us to keep using the iterator even at the end of the iteration. This is depicted in Figure 1 with a shaded box.

The interface of the iterator in state `SOME` must give access to both the `next` and `peek` operations. A tentative type for the iterator in this state is the one satisfying the equation

$$t_{\texttt{SOME}} = \texttt{peek(some}(t_{\texttt{SOME}})) \oplus \texttt{next(reply(int}, t_?))$$

where `peek` replies with a `some` message containing the iterator unchanged, whereas `next` returns the current element of the array being scanned (of type `int`) and the iterator in an updated state. Inspection of Listing 2 reveals that, after a `next` operation, the iterator temporarily moves into state `INIT` and then eventually reaches either state `SOME` or state `NONE`. Therefore, the type $t_?$ exposing the public interface in this unresolved state is obtained as the "intersection" of the interfaces of the two possible states. More precisely, $t_?$ must be a supertype of both $t_{\texttt{NONE}}$ and $t_{\texttt{SOME}}$. It is not difficult to verify that the $\leqslant$-least upper bound of $t_{\texttt{NONE}}$ and $t_{\texttt{SOME}}$ is

$$t_{\texttt{BOTH}} = \texttt{peek(some}(t_{\texttt{SOME}}) \oplus \texttt{none}(t_{\texttt{NONE}}))$$

showing that, when the state of the iterator is uncertain, only `peek` is allowed. Observe also that `peek` has different types depending on whether the state of the iterator is known or not: when the state is known, the type of `peek` is more precise (only `some` or only `none` is sent); when the state is unknown, the type of `peek` is less precise (either `some` or `none` is sent). Subtyping tunes the precision of the types of objects, according to the knowledge of their state. ■

5. Type System

We use type environments for tracking the type of the objects used by processes. A *type environment* Γ is a finite mapping from names to types, written $u_1 : t_1, \ldots, u_n : t_n$ or $\tilde{u} : \tilde{t}$ or $\{u_i : t_i\}_{i \in I}$ as convenient. We write $\emptyset$ for the empty environment, $\mathsf{dom}(\Gamma)$ for the domain of Γ, and Γ_1, Γ_2 for the union of Γ_1 and Γ_2, when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$.

Since each object may be used in different parts of a program according to different interfaces, we need a more flexible environment combination operator than (disjoint) union. The environment in which a process is typed describes how the process uses the objects for which there is a type assignment in the environment. If the *same* object is simultaneously used by two (or more) processes, its type will be the combination (*i.e.*, the product) of all the types it has in the environments used for typing the processes. For example, if some object u is shared by two distinct processes P and Q running in parallel, P uses u according to t and Q uses u according to s, then the parallel composition of P and Q uses u according to $t \otimes s$. If, on the other hand, the object u is used by only one of the two processes, say P, according to t, then it is used according to t also by the parallel composition of P and Q. Formally, we define an operation $\otimes$ for combining type environments, thus:

Definition 5.1 (environment combination)**.** The *combination* of Γ_1 and Γ_2 is the type environment $\Gamma_1 \otimes \Gamma_2$ such that $\mathsf{dom}(\Gamma_1 \otimes \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ defined by:

$$(\Gamma_1 \otimes \Gamma_2)(u) \stackrel{\text{def}}{=} \begin{cases} \Gamma_1(u) & \text{if } u \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2) \\ \Gamma_2(u) & \text{if } u \in \mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1) \\ \Gamma_1(u) \otimes \Gamma_2(u) & \text{otherwise} \end{cases}$$

Many substructural type systems define analogous operators for combining type environments. See for example $+$ in [30] or $\uplus$ in [44].

It is also convenient to extend the subtyping relation to type environments, to ease the application of subsumption. Intuitively, the relation $t \leqslant s$ indicates that an object of type t "has more features" than an object of type s. Similarly, we wish to extend $\leqslant$ to environments so that $\Gamma \leqslant \Delta$ indicates that the environment Γ has more resources with possibly more features than Δ. We must be careful not to introduce in Γ linear resources that are not in Δ, for this would allow processes to ignore objects for which they have obligations. Technically, we allow weakening for non-linear objects only. The extension of $\leqslant$ to type environments is formalized thus:

Definition 5.2 (environment subtyping)**.** We write $\Gamma \leqslant \Delta$ if:

1. $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma)$, and
2. $\Gamma(u) \leqslant \Delta(u)$ for every $u \in \mathsf{dom}(\Delta)$, and
3. $\mathsf{nl}(\Gamma(u))$ for every $u \in \mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Delta)$.

Using environment subtyping, we can express the fact that an environment Γ only contains non-linear resources by checking whether $\Gamma \leqslant \emptyset$ holds. In this case, we write $\mathsf{nl}(\Gamma)$.

With these notions, we can start commenting on the rules of the type system, shown in Table 3. The rules allow deriving various judgments, for processes, molecules, patterns, classes, and solutions.

Rule [T-NULL] states that the idle process is well typed only in an empty environment. Since the idle process does nothing, the absence of linear objects in the environment makes sure that no linear object is left unused. On the other hand, non-linear objects can always be discharged using subsumption [T-SUB], which will be described shortly.

Rule [T-SEND] types message sending $u\,.\,M$, where u is an object and M a molecule of messages. This process is well typed if the type of the object coincides with that of the molecule, which as we will see is just the $\otimes$-composition of the types of the messages in it. Note the use of $\otimes$ in the type environment allowing u to possibly occur in M as the argument of some message.

Rule [T-PAR] types parallel compositions $P_1 \mid P_2$. The rule combines the type environments used for typing P_1 and P_2 to properly keep track of the overall use of the objects shared by the two processes.

Rule [T-OBJECT] types object definitions $\texttt{def}\ a = C\ \texttt{in}\ P$. A type t is guessed for the object a and checked to be appropriate for the class C ("appropriateness" will be discussed along with the typing rules for classes) and assigned to a also for typing P. Note that the class C is checked in an environment that contains only a (that is "self"). That is, the type system forces classes to contain no free names other than the reference to self. In principle this is not a restriction, as we have seen in Example 3.3, although in practice it is desirable to allow for more flexibility. We have made this choice to keep the type system as simple as possible. In fact, the type system would remain sound if we allowed C to access non-linear objects. Allowing C to access linear objects is a much more delicate business that requires non-

Typing rules for processes $\boxed{\Gamma \vdash P}$

[T-NULL]
$$\frac{}{\emptyset \vdash \texttt{null}}$$

[T-SEND]
$$\frac{\Gamma \vdash M :: T}{\Gamma \otimes u : T \vdash u\,.\,M}$$

[T-PAR]
$$\frac{\Gamma_i \vdash P_i \ ^{(i=1,2)}}{\Gamma_1 \otimes \Gamma_2 \vdash P_1 \mid P_2}$$

[T-OBJECT]
$$\frac{a : t \vdash C \qquad \Gamma, a : t \vdash P}{\Gamma \vdash \texttt{def}\ a = C\ \texttt{in}\ P}$$

[T-SUB]
$$\frac{\Delta \vdash P}{\Gamma \vdash P}\ \Gamma \leqslant \Delta$$

Typing rules for molecules $\boxed{\Gamma \vdash M :: T}$

[T-MSG-M]
$$\frac{\mathsf{usable}(\tilde{t})}{\bigotimes_{i=1..n} u_i : t_i \vdash \mathtt{m}(\tilde{u}) :: \mathtt{m}(\tilde{t})}\ \begin{array}{l}\tilde{u} = u_1, \ldots, u_n \\ \tilde{t} = t_1, \ldots, t_n\end{array}$$

[T-COMP-M]
$$\frac{\Gamma_i \vdash M_i :: T_i \ ^{(i=1,2)}}{\Gamma_1 \otimes \Gamma_2 \vdash M_1 \mid M_2 :: T_1 \otimes T_2}$$

Typing rules for patterns $\boxed{\Gamma \vdash J :: T}$

[T-MSG-P]
$$\frac{\mathsf{usable}(\tilde{t})}{\tilde{x} : \tilde{t} \vdash \mathtt{m}(\tilde{x}) :: \mathtt{m}(\tilde{t})}$$

[T-COMP-P]
$$\frac{\Gamma_i \vdash J_i :: T_i \ ^{(i=1,2)}}{\Gamma_1, \Gamma_2 \vdash J_1 \mid J_2 :: T_1 \otimes T_2}$$

Typing rules for classes $\boxed{u : t \vdash C}$

[T-REACTION]
$$\frac{\Gamma \vdash J :: T \qquad \Gamma, a : s \vdash P}{a : t \vdash J \triangleright P}\ \begin{array}{l} t \downarrow T \\ t \leqslant t[T] \otimes s\end{array}$$

[T-CLASS]
$$\frac{a : t \vdash C_i \ ^{(i=1,2)}}{a : t \vdash C_1\ \texttt{or}\ C_2}$$

Typing rules for solutions $\boxed{\vdash \mathscr{D} \Vdash \mathscr{P}}$

[T-DEFINITIONS]
$$\frac{a_i : t_i \vdash C_i \ ^{(i \in I)}}{\{a_i : t_i\}_{i \in I} \vdash \{a_i = C_i\}_{i \in I}}$$

[T-PROCESSES]
$$\frac{\Gamma_i \vdash P_i \ ^{(i \in I)}}{\bigotimes_{i \in I} \Gamma_i \vdash \{P_i\}_{i \in I}}$$

[T-SOLUTION]
$$\frac{\Gamma \vdash \mathscr{D} \qquad \Delta \vdash \mathscr{P}}{\vdash \mathscr{D} \Vdash \mathscr{P}}\ \Gamma \leqslant \Delta$$

Table 3. Typing rules.

trivial reasoning on the sequence of firings of the rules in C; we leave this as a future extension.

Rule [T-SUB] is the subsumption rule, allowing us to enrich the type environment of a process according to Definition 5.2. Intuitively, if P is well typed using the objects described by Δ, then it certainly is well typed in an environment $\Gamma \leqslant \Delta$ where the same objects have more features than those actually used by P. This rule is also useful for rewriting the types in the environment as well as for weakening Δ with non-linear objects.

The typing rules for molecules derive judgments of the form $\Gamma \vdash M :: T$. The environment Γ describes the type of the arguments *sent* along the messages in M. The only remarkable feature is the side condition $\mathsf{usable}(\tilde{t})$ in [T-MSG-M], which requires the arguments of a message to be usable or, at least, discardable. This condition is essential for the soundness of the type system (see Example 6.7).

The typing rules for patterns have the form $\Gamma \vdash J :: T$ and are similar to those for molecules. Recall that patterns occur on the left hand side of reaction rules. In this case, the environment Γ describes the type of the arguments *received* when the pattern matches a molecule in the soup. There is a technical difference between [T-COMP-M] and [T-COMP-P]: the former uses the operator $\otimes$ for combining type environments, as it may happen that the same object is sent as argument in different messages; the latter takes the union of the environments, which are known to have disjoint domains because of the linearity restrictions we have imposed on patterns. Also recall that joined patterns must have disjoint signatures.

Before looking at the typing rules for classes, let us get rid of those for solutions $\mathscr{D} \Vdash \mathscr{P}$, which are essentially unremarkable. Each object definition in $\mathscr{D}$ is typed as in rule [T-OBJECT] and the processes in the multiset $\mathscr{P}$ are typed as if they were all composed in parallel. The two typings are kept consistent by the fact that [T-SOLUTION] uses *related* environments Γ and Δ for both $\mathscr{D}$ and $\mathscr{P}$. The reason why Δ is not exactly Γ is purely technical and accounts for the fact that the subsumption rule [T-SUB] can be applied to the parallel composition of processes $P_1 \mid P_2$, but not after the two processes have been heated and split into the multiset P_1, P_2. More details are provided in [10].

The type system described so far is rather ordinary: the typing rules track the usage of objects, most of the heavy lifting is silently done by subtyping and the $\otimes$ operator. The heart of the type system is [T-REACTION], which verifies that a reaction rule $J \triangleright P$ is appropriate for an object a of type t. The rule determines the type T and bindings Γ of the pattern J and checks that the body P of the rule is well typed in the environment $\Gamma, a : s$. Having a in the environment grants P access to "self". Now, we have to understand which relations should hold among t, T, and s in order for the reaction rule to be safe. In this context "safe" means that:

(1) T describes correctly the type of the received arguments. This is not obvious, because the same tag can be used in messages with arguments of different types (see for example `peek` in Example 3.2) while reduction picks messages solely looking at their tag (Table 2).

(2) By using a according to s, P restores the state of a into one of its valid configurations, described by t. Again this is not obvious, because the only knowledge that P has regarding the state of a comes from the matching of J, which in general is a fraction of all the messages targeted to a at the time of the reaction.

Condition (1) is verifies by a predicate $t \downarrow T$ saying when a given molecule type T is not ambiguous in t:

Definition 5.3 (clear pattern)**.** We say that T is *clear* in t, notation $t \downarrow T$, if $\{S \mid S \otimes R \in [\![t]\!] \wedge \overline{S} = \overline{T}\} = \{T\}$.

In words, $t \downarrow T$ holds if for each valid configuration $S \otimes R$ of t that includes a molecule type S sharing the same signature as T, the molecule type is exactly T. In addition, there must be a valid configuration of t that includes T. This implies that, whenever $t \downarrow T$ holds, t is usable.

For example, take $t \stackrel{\text{def}}{=} (\mathtt{A} \otimes \mathtt{m(int)}) \oplus (\mathtt{B} \otimes \mathtt{m(real)})$ and observe that the argument of message `m` has different types depending on whether the state of the object is `A` or `B`. Then, neither $t \downarrow \mathtt{m(int)}$ nor $t \downarrow \mathtt{m(real)}$ holds, for matching an `m`-tagged message does not provide enough information for deducing the type of its argument. However, both $t \downarrow \mathtt{A} \otimes \mathtt{m(int)}$ and $t \downarrow \mathtt{B} \otimes \mathtt{m(real)}$ do hold, since in these cases the signature of the matched molecule disambiguates the type of `m`'s argument.

Remark 5.4. Suppose $t = (\mathtt{m(foo)} \otimes \mathtt{m(bar)}) \oplus \mathbb{1}$ and that the reaction $J \triangleright x\,\mathtt{.foo} \mid y\,\mathtt{.bar}$ where $J = \mathtt{m}(x) \mid \mathtt{m}(y)$ is allowed, despite `m` occurs twice in J. We have

$$x : \mathtt{foo}, y : \mathtt{bar} \vdash J :: T$$

where $T = \mathtt{m(foo)} \otimes \mathtt{m(bar)}$. Now $t \downarrow T$ does hold, because t has only one valid configuration with the same signature as T. However, there is no guarantee that, once J matches a molecule, x is actually bound to the object of type `foo` and y is actually bound to the object of type `bar`, and not vice versa. For this reason, pattern linearity is a key restriction in our type system, where messages with the same tag can have arguments with different types. ■

To find guidance for verifying condition (2) it helps recalling the chemical interpretation of the Objective Join Calculus: the effect of a reaction $J \triangleright P$ where J has type T and P uses the object according to s is to *consume* a molecule of messages of type T and to *produce* molecules according to type s. The reaction is safe if the overall balance between what is consumed and what is produced preserves the object's configuration as one that is described by its type t. Formally, this is expressed by the side condition $t \leqslant t[T] \otimes s$, where the type $t[T]$ represents the "residual" of t after a molecule with type T (the pattern of the reaction rule) has been removed; such residual is combined (in the sense of $\otimes$)

with s, which is what P sends to the object; the resulting type $t[T] \otimes s$ is compatible with the object's type t if it is a supertype of t. The type residual operator is defined thus:

Definition 5.5 (type residual)**.** The *residual* of t with respect to $\mathtt{M}$, written $t[\mathtt{M}]$, is inductively defined thus:

$$\begin{aligned}
\mathbb{0}[\mathtt{M}] &= \mathbb{1}[\mathtt{M}] = \mathbb{0} \\
\mathtt{m}(\tilde{t})[\mathtt{m}'(\tilde{s})] &= \mathbb{0} && \text{if } \mathtt{m} \neq \mathtt{m}' \\
\mathtt{m}(\tilde{t})[\mathtt{m}(\tilde{s})] &= \mathbb{1} \\
(t \oplus s)[\mathtt{M}] &= t[\mathtt{M}] \oplus s[\mathtt{M}] \\
(t \otimes s)[\mathtt{M}] &= (t[\mathtt{M}] \otimes s) \oplus (t \otimes s[\mathtt{M}]) \\
(*t)[\mathtt{M}] &= t[\mathtt{M}] \otimes *t
\end{aligned}$$

We extend the residual to molecule types in the obvious way, that is $t[\mathbb{1}] = t$ and $t[\mathtt{M} \otimes T] = t[\mathtt{M}][T]$.

Note that the type residual operator (Definition 5.5) is nothing but *Brzozowski derivative* [6, 8] adapted to a commutative Kleene algebra over message types.

To better illustrate the side condition, we work out some examples in which we consider different objects a of type t and we write $T \triangleright s$ for denoting a reaction $J \triangleright P$ where J has type T and P is typed in an environment that includes $a : s$. We will say that $T \triangleright s$ is valid or invalid depending on whether the condition holds or not.

- If $t \stackrel{\text{def}}{=} \mathtt{A} \oplus (\mathtt{B} \otimes \mathtt{m})$, then $\mathtt{A} \triangleright \mathtt{B} \otimes \mathtt{m}$ and $\mathtt{B} \otimes \mathtt{m} \triangleright \mathtt{A}$ are valid but $\mathtt{B} \triangleright \mathtt{A}$ is not. For example, we have $t[\mathtt{B}] = \mathtt{m}$ and $t \not\leqslant \mathtt{m} \otimes \mathtt{A}$. When in state $\mathtt{B}$, there is also a message $\mathtt{m}$ that is forbidden in state $\mathtt{A}$.
- If $t \stackrel{\text{def}}{=} (\mathtt{A} \otimes \mathtt{m}) \oplus \big(\mathtt{B} \otimes (\mathbb{1} \oplus \mathtt{m})\big)$, then $\mathtt{A} \triangleright \mathtt{B}$ is valid but $\mathtt{B} \triangleright \mathtt{A}$ is not. We have $t[\mathtt{B}] = \mathbb{1} \oplus \mathtt{m}$ and $t \not\leqslant (\mathbb{1} \oplus \mathtt{m}) \otimes \mathtt{A}$. In general, the transition from a state in which a message is linear ($\mathtt{m}$) to another where the message is not linear ($\mathbb{1} \oplus \mathtt{m}$) cannot be reversed, because the object may have been discarded or aliased.
- If $t \stackrel{\text{def}}{=} \mathtt{A} \oplus (\mathtt{B} \otimes *\mathtt{foo}) \oplus (\mathtt{C} \otimes *\mathtt{foo} \otimes *\mathtt{bar})$, then $\mathtt{A} \triangleright \mathtt{B}$ and $\mathtt{B} \triangleright \mathtt{C}$ are valid, but neither $\mathtt{B} \triangleright \mathtt{A}$ nor $\mathtt{C} \triangleright \mathtt{B}$ is. It is unsafe for the object to move from state $\mathtt{C}$ to state $\mathtt{B}$ because there could be residual $\mathtt{bar}$ messages not allowed in state $\mathtt{B}$. In general, non-linear messages such as $\mathtt{foo}$ and $\mathtt{bar}$ can only accumulate monotonically across state transitions.
- If $t \stackrel{\text{def}}{=} (\mathtt{A} \otimes \mathtt{m}(\mathtt{int})) \oplus (\mathtt{B} \otimes \mathtt{m}(\mathtt{real}))$, then $\mathtt{A} \triangleright \mathtt{B}$ is valid, but $\mathtt{B} \triangleright \mathtt{A}$ is not. Indeed $t[\mathtt{B}] = \mathtt{m}(\mathtt{real})$ and $t \not\leqslant \mathtt{m}(\mathtt{real}) \otimes \mathtt{A}$. The transition $\mathtt{A} \triangleright \mathtt{B}$ is safe because the $\mathtt{int}$ argument of message $\mathtt{m}$ in state $\mathtt{A}$ can be subsumed to $\mathtt{real}$ in state $\mathtt{B}$, but not vice versa.

Example 5.6 (lock)**.** We illustrate the type system at work showing that the two reactions of the lock (lines 2–3 in Listing 3) are well typed using t_{lock}, the types $t_{\mathtt{ACQUIRE}}$ and $t_{\mathtt{RELEASE}}$ of Example 4.6, and also $t_{\mathtt{REP}} \stackrel{\text{def}}{=} \mathtt{reply}(t_{\mathtt{RELEASE}})$. Consider the first reaction; for its pattern we derive

$$\mathtt{r} : t_{\mathtt{REP}} \vdash \mathtt{FREE} \mid \mathtt{acquire(r)} :: T$$

where $T \stackrel{\text{def}}{=} \mathtt{FREE} \otimes \mathtt{acquire}(t_{\mathtt{REP}})$. Let $s \stackrel{\text{def}}{=} \mathtt{BUSY} \otimes t_{\mathtt{RELEASE}}$, then for the body of the reaction we derive

$$\dfrac{\dfrac{}{\mathtt{o} : \mathtt{BUSY} \vdash \mathtt{o.BUSY}} \qquad \dfrac{}{\mathtt{o} : t_{\mathtt{RELEASE}} \vdash \mathtt{reply(o)} :: t_{\mathtt{REP}}}}{\mathtt{r} : t_{\mathtt{REP}}, \mathtt{o} : t_{\mathtt{RELEASE}} \vdash \mathtt{r.reply(o)}}$$
$$\mathtt{r} : t_{\mathtt{REP}}, \mathtt{o} : s \vdash \mathtt{o.BUSY} \mid \mathtt{r.reply(o)}$$

Now $t_{lock} \downarrow T$ holds and furthermore

$$t_{lock} \leqslant t_{lock}[T] \otimes s = t_{\mathtt{ACQUIRE}} \otimes \mathtt{BUSY} \otimes t_{\mathtt{RELEASE}}$$

hence the side conditions of [T-REACTION] are satisfied. For the pattern in the second reaction we derive

$$\vdash \mathtt{BUSY} \mid \mathtt{release} :: \mathtt{BUSY} \otimes \mathtt{release}$$

and it is easy to see that the body of the reaction is also well typed. Now, we have $t_{lock} \downarrow \mathtt{BUSY} \otimes \mathtt{release}$ and

$$t_{lock} \leqslant t_{lock}[\mathtt{BUSY} \otimes \mathtt{release}] \otimes \mathtt{FREE} \simeq t_{\mathtt{ACQUIRE}} \otimes \mathtt{FREE}$$

so the side conditions of [T-REACTION] are again satisfied, this time taking $s \stackrel{\text{def}}{=} \mathtt{FREE}$. ■

Example 5.7 (iterator)**.** We conclude the typing of the array iterator in Example 3.2 (Listing 2). By composing the public interfaces defined in Example 4.7, we can define the type of the iterator object $\mathtt{o}$ as follows:

$$\begin{aligned}
t_{iter} \stackrel{\text{def}}{=} &(\mathtt{INIT}(\mathtt{int[]}, \mathtt{int}) \otimes t_{\mathtt{BOTH}}) \\
&\oplus (\mathtt{SOME}(\mathtt{int[]}, \mathtt{int}) \otimes t_{\mathtt{SOME}}) \\
&\oplus (\mathtt{NONE} \otimes t_{\mathtt{NONE}})
\end{aligned}$$

Notice that t_{iter} is obtained as a disjunction of three types, each corresponding to a pair encoding a possible state and the public interface of the iterator in that state.

In order to check the typing of the definition of the object $\mathtt{o}$ in Listing 2, we have to check four reactions; we just discuss two of them and, for readability, we only consider message tags omitting argument types. The first reaction $\mathtt{INIT} \triangleright \mathtt{SOME} \oplus \mathtt{NONE}$ is valid since $t_{iter}[\mathtt{INIT}] = t_{\mathtt{BOTH}}$ and

$$\begin{aligned}
t_{iter} &\leqslant (\mathtt{SOME} \oplus \mathtt{NONE}) \otimes t_{\mathtt{BOTH}} \\
&\simeq (\mathtt{SOME} \otimes t_{\mathtt{BOTH}}) \oplus (\mathtt{NONE} \otimes t_{\mathtt{BOTH}})
\end{aligned}$$

because $t_{\mathtt{SOME}} \leqslant t_{\mathtt{BOTH}}$ and $t_{\mathtt{NONE}} \leqslant t_{\mathtt{BOTH}}$ as we have argued in Example 4.7. The reaction $\mathtt{SOME} \otimes \mathtt{next} \triangleright \mathtt{INIT} \otimes t_{\mathtt{BOTH}}$ is also valid since $t_{iter}[\mathtt{SOME} \otimes \mathtt{next}] = \mathbb{1}$ and now

$$t_{iter} \leqslant \mathbb{1} \otimes \mathtt{INIT} \otimes t_{\mathtt{BOTH}} \simeq \mathtt{INIT} \otimes t_{\mathtt{BOTH}}$$

Observe that the code in Listing 2 does not contain any reaction involving both the state $\mathtt{INIT}$ *and* the operation $\mathtt{peek}$, since the iterator in state $\mathtt{INIT}$ eventually moves into either state $\mathtt{SOME}$ or $\mathtt{NONE}$; nevertheless t_{iter} exposes the interface $t_{\mathtt{BOTH}}$ while in state $\mathtt{INIT}$, instead of the empty interface. This is because, in lines 6 and 9, a reference $\mathtt{o}$

```
1 def Channel = new(c) ▷
2   def o =
3       LE    | lsend(v,c) ▷ o.LF(v) | c.reply(o)
4   or LF(v) | rrecv(c)   ▷ o.LE    | c.reply(v,o)
5   or RE    | rsend(v,c) ▷ o.RF(v) | c.reply(o)
6   or RF(v) | lrecv(c)   ▷ o.RE    | c.reply(v,o)
7   in o.LE | o.RE | c.reply(o)
8 in ···
```

Listing 5. Full-duplex channel.

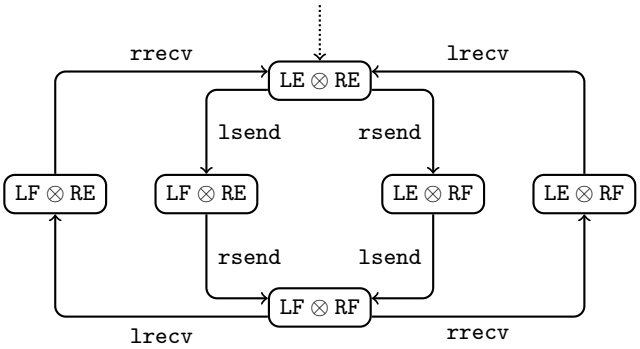

Figure 2. Transition diagram of the full-duplex channel.

to the iterator is returned to the caller while the iterator is moving to state `INIT`. Such reference could be used by a quick caller to send a `peek` message to the iterator while the iterator is still in the transient state `INIT`, and this requires `INIT` $\otimes$ `peek` to be a valid configuration of t_{iter}.

It is possible to make sure that the reference `o` returns to the caller only once the iterator has moved away from state `INIT`, by reshaping `INIT` into a synchronous operation. ■

Example 5.8 (full-duplex channel)**.** Listing 5 shows the modeling of a bidirectional, full-duplex channel for connecting two peer processes, called "left" and "right" and identified by a letter $p \in \{\mathtt{l}, \mathtt{r}\}$. The channel provides two pairs of operations p`send` and p`recv` used by peer p for sending and receiving messages. The state of the channel is modeled by two 1-place buffers, one for each peer. For the left peer, `LE` represents the Empty buffer and `LF`(v) the Full buffer with a value v. Tags `RE` and `RF` are used for representing the buffer of the right peer in a similar way. Observe that each buffer is either empty or full, but the two buffers coexist and can change state independently. This means that `LE` and `LF` are *or-states*, and so are `RE` and `RF`: on the contrary, Lx and Ry are *and-states*. This will be reflected in the type of the channel, where different states of the same buffer are combined by $\oplus$, whereas states of different buffers are combined by $\otimes$.

We want to enforce a usage protocol of the full-duplex channel such that each peer p alternates send and receive operations. In this way, the p`send` of peer p fills the corresponding buffer and enables the $\overline{p}$`recv` of the other peer $\overline{p}$, but only after $\overline{p}$ has sent its own message. Figure 2 depicts the transition diagram of the full-duplex channel used according to this protocol. The interface of the channel from the viewpoint of p is described by the type t_{ps} defined by

$$\begin{aligned} t_{ps} &= p\mathtt{send}(\mathtt{int}, \mathtt{reply}(t_{pr})) \\ t_{pr} &= p\mathtt{recv}(\mathtt{reply}(\mathtt{int}, t_{ps})) \end{aligned}$$

The types of the interfaces are combined with state message types to form the type of the channel as follows

$$\begin{aligned} t_{chan} \stackrel{\text{def}}{=} &(\mathtt{LE} \otimes \mathtt{RE} \otimes t_{\mathtt{ls}} \otimes t_{\mathtt{rs}}) \oplus (\mathtt{LF} \otimes \mathtt{RF} \otimes t_{\mathtt{lr}} \otimes t_{\mathtt{rr}}) \\ \oplus &(\mathtt{LF} \otimes \mathtt{RE} \otimes t_{\mathtt{lr}} \otimes t_{\mathtt{rs}}) \oplus (\mathtt{LE} \otimes \mathtt{RF} \otimes t_{\mathtt{ls}} \otimes t_{\mathtt{rr}}) \\ \oplus &(\mathtt{LF} \otimes \mathtt{RE} \otimes t_{\mathtt{ls}} \otimes t_{\mathtt{rr}}) \oplus (\mathtt{LE} \otimes \mathtt{RF} \otimes t_{\mathtt{lr}} \otimes t_{\mathtt{rs}}) \end{aligned}$$

where we have elided the type of values in the buffers.

Inspection of t_{chan} reveals that the reference `o` returned on line 7 has type $t_{\mathtt{ls}} \otimes t_{\mathtt{rs}}$, that is the composition of the two public interfaces of the channel, each corresponding to one of the peers. Therefore, the *same* channel object can be used by two parallel processes, according to these two types, as illustrated by the code snippet below:

```
let c = Channel.new in        (* c : t_ls ⊗ t_rs *)
{ let c   = c.lsend(1) in     (* c : t_lr *)
  let v,c = c.lrecv in ...    (* c : t_ls *)
| let c   = c.rsend(2) in     (* c : t_rr *)
  let v,c = c.rrecv in ... }  (* c : t_rs *)
```

The internal state of the full-duplex channel is the *combination* of distinct messages Lx and Ry that are consumed and produced *concurrently* by the users of the channel. In particular, each reaction rule in Listing 5 changes only *part* of the channel's state, leaving the rest unchanged. The last side condition of rule [T-REACTION] verifies that such partial change maintains the channel's overall state in one of the configurations described by t_{chan}. The interested reader can verify that each reaction rule is indeed well typed with respect to t_{chan}.

As a final consideration, the fact that t_{chan} and the diagram in Figure 2 list 6 configurations (instead of the 4 corresponding to all possible combinations of Lx and Ry) suggests that the interface of the channel depends not only on its *current state* (encoded as a pair L$x \otimes$ Ry) but also on its *past history*. For instance, in the two states identified by the combination of messages `LF` $\otimes$ `RE`, the peer `l` has produced its own message while the buffer of peer `r` is empty. But this can be either because `r` has not produced its own message yet, or because `r` has indeed produced the message, and peer `l` has already received it. ■

6. Properties of Well-Typed Processes

In this section we prove a few properties enjoyed by well-typed processes. To begin with, we state a completely standard, yet fundamental result showing that typing is preserved under heating, cooling, and reductions.

Theorem 6.1 (subject reduction). *If* $\vdash \mathscr{D} \Vdash \mathscr{P}$ *and*

$$\mathscr{D} \Vdash \mathscr{P} \; \mathscr{R} \; \mathscr{D}' \Vdash \mathscr{P}'$$

where $\mathscr{R} \in \{\rightharpoonup, \rightharpoondown, \to\}$, *then* $\vdash \mathscr{D}' \Vdash \mathscr{P}'$.

Theorem 6.1 is key for the next results, since it assures that the properties enjoyed by well-typed processes are invariant under arbitrarily long process reductions.

The first proper soundness result states that a well-typed process respects the prohibitions expressed by the types of the objects it manipulates. We formulate this property stating that if a well-typed solution contains messages $\mathtt{m}_1, \ldots, \mathtt{m}_n$ targeted at some object a and a has type t, then there is a valid configuration of t that includes at least all the $\mathtt{m}_i$, and possibly more messages.

Theorem 6.2 (respected prohibitions). *If*

$$\Gamma, a : t \vdash P \mid a.\mathtt{m}_1(\tilde{c}_1) \mid \cdots \mid a.\mathtt{m}_n(\tilde{c}_n)\ ,$$

then there exist S *and* $\tilde{s}_i$ *such that* $t \leqslant S \otimes \bigotimes_{i=1..n} \mathtt{m}_i(\tilde{s}_i)$.

The theorem can be rephrased in terms of prohibitions as follows: if the type of an object prohibits invocation of a particular method when the object is in some particular state, then there is no well-typed soup of processes containing pending invocations to that method when the object is in that state. To illustrate, consider the lock object in Listing 1. The type t_{lock} of the lock we have defined in Section 2 prohibits invocation of method `release` when the lock is in state `FREE`. Also recall that the lock being in the `FREE` state is identified by the presence of the `o.FREE` molecule in the solution. Then, the following judgment is *not* derivable

$$\Gamma, \mathtt{o} : t_{lock} \vdash P \mid \mathtt{o.FREE} \mid \mathtt{o.release}$$

Similarly, the type t_{lock} states that, when in state `BUSY`, there can be exactly one pending invocation to `release`. So,

$$\Gamma, \mathtt{o} : t_{lock} \vdash P \mid \mathtt{o.BUSY} \mid \mathtt{o.release} \mid \mathtt{o.release}$$

is another judgment that cannot be derived. Remarkably, we can infer a great deal of information regarding the state of an object by solely looking at its type, knowing virtually nothing about the rest of the (well-typed) program. For instance, no soup containing both a `FREE` and a `BUSY` message simultaneously targeted to the same lock is well typed, meaning that the state of every lock is always uniquely determined.

The second soundness result states that a well-typed process fulfills all the obligations with respect to the objects it owns. More precisely, that if a process P is typed in an environment that contains a linear object a, that is an object whose type mandates the (eventual) invocation of a particular method, then a cannot be discarded by P, but must be held by P and used according to its type.

Theorem 6.3 (weakly fulfilled obligations). *If* $\Gamma \vdash P$ *and* $a \in \mathsf{dom}(\Gamma)$ *and* $\mathsf{lin}(\Gamma(a))$, *then* $a \in \mathsf{fn}(P)$.

```
1 def Lock = ...                    (* see Listing 3 *)
2 in let lock  = Lock.new           (* lock  : t_ACQUIRE *)
3 in let lock1 = lock.acquire       (* lock1 : t_RELEASE *)
4 in let lock2 = lock.acquire       (* lock2 : t_RELEASE *)
5 in lock1.release | lock2.release
```

Listing 6. Modeling of a deadlock.

Another way of reading this theorem is that well-typed processes can only drop non-linear objects, namely objects for which they have no pending obligations. For example, since t_{lock} mandates the invocation of method `release` once the lock has been acquired, omitting the `f.release` from line 5 in Listing 4 would result into an ill-typed philosopher.

We have labeled Theorem 6.3 "weak" obligation fulfillment because the property may indeed look weaker than desirable. One would probably expect a stronger property saying that every method that must be invoked *is* eventually invoked. Such stronger property, which is in fact a *liveness* property, is however quite subtle to characterize and hard to enforce with a type system. In particular, it would require well-typed processes to be free from both deadlocks and livelocks, which is something well beyond the capabilities of the type system we have presented in Section 5. The next two examples illustrate why this is the case.

Example 6.4 (deadlock). Assuming a `Lock` class defined as in Listing 3, the code in Listing 6 attempts at acquiring the *same* lock twice, resulting in a deadlock. In particular, a new `lock` is created on line 2 and used twice on lines 3 and 4 for acquisition. This is possible because of the relation $t_{\texttt{ACQUIRE}} \simeq t_{\texttt{ACQUIRE}} \otimes t_{\texttt{ACQUIRE}}$. Clearly, only the first acquisition succeeds, and the program blocks while performing the second one. Note that the program is well typed, as both `lock1` and `lock2`, which have a linear type, are used in line 5 for releasing the lock, according to the lock protocol. However, such "usage" is merely syntactic, for neither of the two `release` messages on line 5 will ever be received, and the lock will never be released. Note that static deadlock detection is undecidable in general and non-trivial to approximate. In the above example, for instance, it would require understanding that the `acquire` method is a *blocking* one (this information cannot be inferred merely from the type of `Lock`) and that it is the *same* lock being acquired twice, in a fragment of sequential code (this is easy to detect in the above example, where `lock` syntactically occurs twice, but in general the code could invoke the `acquire` method on distinct variables that are eventually instantiated with the same reference to `lock`). ■

Example 6.5 (livelock). There is one trivial way to honor all pending obligations (as by Theorem 6.3), namely postponing them forever. For example, let

$$\mathit{forever}(u) \stackrel{\text{def}}{=} \mathtt{def}\ c = \mathtt{m}(x) \triangleright c.\mathtt{m}(x)\ \mathtt{in}\ c.\mathtt{m}(u)$$

where c is a fresh name. The judgment $a : t \vdash \mathit{forever}(a)$ is derivable for any t such that $\mathsf{usable}(t)$. In particular, t may be linear, and yet $\mathit{forever}(a)$ never invokes any method on a. Although $\mathit{forever}(a)$ fools the type system into believing that all pending obligations on a have been honored, processes like $\mathit{forever}(a)$ are sufficiently contrived to be rarely found in actual code. In other words, we claim that Theorem 6.3 provides practically useful guarantees about the actual use of objects with non-linear types. ■

Finally, we draw the attention on a general property of the type system that is key for proving Theorem 6.2:

Lemma 6.6. *There exist no* Γ *and* P *such that* $\Gamma, u : \mathbb{0} \vdash P$.

This property states that there is no well-typed process that can hold an unusable object. The result may look obvious, but it has important consequences: we have remarked the role of subtyping for deducing the interface of objects with uncertain state. For instance, $t_{\texttt{BOTH}}$ (Example 4.7) is obtained as the *least upper bound* of $t_{\texttt{NONE}}$ and $t_{\texttt{SOME}}$. Since $\mathbb{0}$ is the top type, the least upper bound of two (or more) types *always* exists, but it can be $\mathbb{0}$. For example, had we forgotten to equip the iterator with a `peek` operation in state `SOME` (line 7 of Listing 2), $t_{\texttt{BOTH}}$ would be $\mathbb{0}$ and the iterator would be essentially unusable. Lemma 6.6 tells us that the type system detects such mistakes.

Example 6.7. The side condition in rule [T-MSG-M] requires the arguments of a message to have a usable type. If this condition were not enforced, the following derivation would be legal and Theorem 6.2 would not hold:

$$\dfrac{\dfrac{\dfrac{\vdots}{a : \mathbb{0} \vdash \mathit{forever}(a)} \qquad \dfrac{\dfrac{}{\vdash \texttt{bar} :: \texttt{bar}}\,[\text{T-MSG-M}]}{a : \texttt{bar} \vdash a.\texttt{bar}}\,[\text{T-SEND}]}{a : \mathbb{0} \otimes \texttt{bar} \vdash \mathit{forever}(a) \mid a.\texttt{bar}}\,[\text{T-PAR}]}{a : \texttt{foo} \vdash \mathit{forever}(a) \mid a.\texttt{bar}}\,[\text{T-SUB}]$$

Since $\texttt{foo} \leqslant \mathbb{0} \otimes \texttt{bar} \simeq \mathbb{0}$, the subsumption rule could be used for allowing spurious method invocations (`bar`) knowing that these would be absorbed by $\mathbb{0}$ types in other parts of the derivation. Note that the message invocation in $\mathit{forever}(a)$ sends an object with type $\mathbb{0}$. ■

7. Implementation Aspects

The implementation of our approach to TSOP involves design and implementation aspects covering both the *program level* (the runtime support for handling messages, matching join patterns, firing reactions) and the *type level* (the compile-time support for type checking). We give an account of such aspects in this section.

Concerning the program level, our framework relies on a standard formulation of the Join Calculus, for which there exist standalone, embedded, and library implementations: native support for join patterns is provided in JoCaml [20], Join Java [28], and in Cω [3, 33], among others; library implementations of join patterns are available for C# [42, 49], Visual Basic [43], and Scala [24]. Both native and library implementations of join patterns have pros and cons. Natively supported join patterns allow for specific optimizations [31] and analysis techniques [18, 37], but they are currently available only for niche programming langauges that enjoy limited popularity. Library implementations of join patterns are (or can be made) available for all mainstream programming languages and therefore integrate more easily with existing code and development environments, but they might be constrained by the syntax and typing discipline of the host language. Nonetheless, carefully crafted implementations can perform and scale remarkably well [49].

To illustrate the applicability of our approach, Listing 7 presents the full-duplex channel (Example 5.8) implemented using the Scala Joins library [24], which defines Join's control structures on top of the standard Scala language.[1] The code defining the `Channel` class has a straightforward correspondence with its formal counterpart. The class consists of a set of *event declarations* specifying the messages that can be targeted to instances of the class (lines 3–10), the *reaction rules* specifying the behavior of instances of the class (lines 11–20), as well as the initial state of each instance (lines 21–22). The main program creates a channel that is shared by two asynchronous processes that exchange integers and strings in full-duplex (lines 37–39). The messages representing the state of the channel (lines 3–6) are `private` to enforce encapsulation of the state. The public interface (lines 7–10) is represented by synchronous events, which means that invoking a `Channel`'s public operation (lines 27–28 and 31–32) suspends the calling thread until the matching reaction has fired and a result has been returned (lines 13,15,17,19). In the formal model, all message sends are asynchronous and sequentiality is encoded with explicit continuations (see the reference `c` in Listing 5). The reaction rules that govern the channel behavior are defined by means of a call to the `join` method (inherited from the `Joins` superclass in the Scala Joins library) that takes as a parameter a partial function encoding the join patterns in terms of pattern matching. Pattern composition is then achieved by means of the `and` combinator, whose definition exploits Scala's extractors and extensible pattern matching. Note that `Channel` is parametric in the types `L` and `R` of the messages exchanged over the full-duplex channel. This possibility, not accounted for in the formal presentation of the type system (Section 5), comes for free thanks to Scala's support for generics.

Imposing the typing discipline that we have described in Section 5 is more challenging to put into practice since this requires the implementation of a substructural type system that makes use of unconventional behavioral connectives. When using the Scala Joins library, the programmer can only

[1] The example has been written and tested using a variant of Scala Joins 0.4 that has been patched to make it compatible with Scala 2.11.6.

```
class Channel[L,R] extends Joins {
  // MESSAGES //
  private object LE extends NullaryAsyncEvent
  private object RE extends NullaryAsyncEvent
  private object LF extends AsyncEvent[L]
  private object RF extends AsyncEvent[R]
  object lsend extends SyncEvent[Unit,L]
  object rsend extends SyncEvent[Unit,R]
  object lrecv extends NullarySyncEvent[R]
  object rrecv extends NullarySyncEvent[L]
  join { // REACTION RULES //
    case LE()  and lsend(v) => LF(v)
                               lsend reply {}
    case LF(v) and rrecv()  => LE()
                               rrecv reply v
    case RE()  and rsend(v) => RF(v)
                               rsend reply {}
    case RF(v) and lrecv()  => RE()
                               lrecv reply v
  }
  LE() // INITIALIZATION //
  RE()
}

class Process(chan : Channel[Int,String]) {
  def runRight(v : String) : Unit = {
    println("Right sends " + v); chan.rsend(v)
    println("Right receives " + chan.rrecv())
    runRight(v + "*") }
  def runLeft(v : Int) : Unit = {
    println("Left sends " + v); chan.lsend(v)
    println("Left receives " + chan.lrecv())
    runLeft(v + 1) }
}

object TestChannel extends App {
  val chan = new Channel[Int,String]
  Future { new Process(chan).runLeft(1930) }
  Future { new Process(chan).runRight("Pluto") }
  Thread.sleep(5000000)
}
```

Listing 7. The full-duplex channel in Scala Joins [24].

rely on the native Scala type checker, which can verify that programs comply with the *interface* of the objects they use, but not necessarily with their *protocol*. The compiler can detect if a message of the wrong type is sent to an object or if a message is sent to an object that does not expose that message in its interface, but it cannot verify whether messages are sent in a particular order, or if a program fulfils the obligations with respect to the objects it uses.

We envision two ways of implementing the typing discipline advocated in this paper: the first one is to develop a TSOP-aware programming language, possibly integrated with one or more host languages, in the style of Plaid [1, 48]; the second one is to superimpose our type system to that of an existing programming language, augmented with a DSL for TSOP. The first approach would grant us complete control over the type checker and would make it possible to take full advantage of typing information, for example to minimize the amount and nature of runtime checks concerning typestates. A major downside is that the language would likely enjoy limited popularity. The second approach has been pursued, for example, in Mungo,[2] a Java front-end that implements a behavioral type system for Java objects exposing dynamically changing interfaces. The key idea in Mungo is to have a pre-processing phase that analyzes the code using a typestate-sensitive type checker. If this phase is passed, the program is handed over to the standard Java compiler. The stratification of this architecture could favor the integration of our framework with a wider range of programming languages and development environments, possibly at the cost of some compromises in the use of type information for checking protocol compliance and optimal code generation.

8. Related Work

Typestate-Oriented Programming. In [14] class states are represented as invariants describing predicates over fields. They support verification in presence of inheritance and depend on a classification of references as not aliased or possibly aliased. This approach is refined in [1, 5] with a flexible access permission system that permits state changes even in the presence of aliasing. Shared access permissions have been investigated in a concurrent framework in [45, 46], but their integration with the typestate mechanism has not been considered in these works. In Plaid [48], the typestate of an object directly corresponds to its class, and that class can change dynamically. Plaid supports the major state modeling features of Statecharts: state hierarchy, *or-states*, and *and-states*, allowing states dimensions to change independently.

The foundations of Plaid and, in general, of TSOP are formally studied in [21] by means of a nominal object-oriented language with mutable state and a native notion of typestate change. The language is also equipped with a permission-based type system integrated with a gradual typing mechanism that combines static and dynamic checking. Progress and type preservation properties are formally proved.

To the best of our knowledge, TSOP has been investigated in a concurrent setting only in [11] and marginally in [23]. Damiani *et al.* [11] develop a type and effect system for a Java-like language to trace how the execution of a method changes the state of the receiver object. To forbid access to fields that are not available in the current object's state, only direct invocations of methods on `this` can change the state of the current object. Since each class method is synchronized, two concurrent threads cannot simultaneously execute in the same object. Our approach relaxes such restrictions. For instance, the full-duplex channel (Example 5.8) can be used in true concurrency by the two peers, and each is stat-

[2] `http://www.dcs.gla.ac.uk/research/mungo/`

ically guaranteed to comply with (its view of) the channel protocol. Gay *et al.* [23] study an integration of typestate and session types targeting distributed objects. The focus of their work is on the modularization of sessions across different methods rather than on typestates themselves. In fact, the work rests on the assumption that *non-uniform* objects (those whose interface changes with time) must be used linearly.

Concurrent Objects. In the actor model [26] messages (asynchronously) received by objects are handled by an internal single-threaded control which can dynamically change its behaviour, thereby changing the object/actor's state. In SCOOP [35, 50] each object is associated with a handler thread and the client threads wishing to send requests to the object must explicitly register this desire by using the `separate` construct. Unlike actors, SCOOP's threads have more control over the order in which the receiver processes messages: the messages from a single separate block are processed in order, without any interleaving. In addition, SCOOP allows pre/postcondition reasoning in a concurrent setting: before executing a method, the executing processor waits until the precondition is satisfied while each postcondition clause is evaluated individually and asynchronously. Pre/post conditions are reminiscent of state-sensitive operations distinctive of TSOP.

Behavioral description of objects with dynamically changing interfaces, sometimes called *active* or *non-uniform* objects, have been studied for more than two decades. Among the seminal works is [36], where Nierstrasz proposes finite-state automata for describing object protocols and failure semantics for characterizing the subtyping relation. Type-theoretic approaches based on similar notions have been subsequently studied for various object calculi and type languages, with varying degrees of ensured properties, for instance in [34, 38–41]. Aside from the fact that none of these works is based on the Join Calculus or addresses TSOP explicitly, all of these type systems are biased towards the description of *operations* rather than state. In [34] and [41], types are variants of process algebras built on actions standing for method invocations. Choices and "parallel" composition roughly correspond to our $\oplus$ and $\otimes$ operators, and sequential composition is used to express the dynamic change of an object's interface. We model this aspect using explicit continuation passing, in the style of [12, 22]. By contrast, the type systems in [38–40] are based on the decoration of object types with *tokens* akin to typestates, in the sense that they can be used to represent in an abstract form the internal state of object that affects the available operations. Tokens themselves can be annotated in such a way so as to express obligations on the use of objects.

Session Types. Communication protocols over channels, as opposed to object protocols, can be described using *session types* [27]. Our use of explicit continuations for describing structured behaviors has been inspired by the encoding of session types into linear channel types [12]. Our type system uses essentially the same technique, except that continuations are objects instead of channels. Continuations are convenient also when types account for structured protocols, to describe the effect of functions on channels [22].

Types for the Join Calculus. Our language of behavioral types is an original contribution of this paper and is also the first behavioral type theory for the Objective Join Calculus. Other type systems for the Join Calculus have been formally investigated in [18, 37] and for the Objective Join Calculus in [19]. [18] discusses an extension of ML-style polymorphism to the Join Calculus and addresses the issues that arise when polymorphic channels are joined in the same pattern; [37] presents a typing discipline to reason on the scope within which channels are allowed to be used, so as enforce a form of encapsulation. These two works are based on the original formulation of the Join Calculus [16], where a join definition introduces a bunch of *channel names* all at once. In particular, objects are modeled indirectly as the set of the operations they support, each operation being represented by a distinct channel. Since types are associated with channels, it is then difficult if at all possible to describe and reason about the overall behavior of objects. The seminal paper on the Objective Join Calculus [19] also introduces a type system for establishing basic safety and privacy properties. In particular, it distinguishes between *public* and *private* messages, the latter ones being used for encoding the internal state of objects. Once again, privacy information is associated with the single messages and object types solely specify their interface, but not their protocol. In our type system, the separation between public and private messages stems from the fact that distinct references to the same object may be given different types eventually combined using $\otimes$.

To the best of our knowledge, [7] is the only work that describes a form of analysis on join patterns that takes object behaviors into account and therefore that is remotely related to our type system. However, [7] is based on a control-flow analysis rather than types and its aim is to optimize code generated by join definitions, for instance detecting that some channels (called *signals* in [7]) never escape the scope of the object in which they are defined and that their associated message queue always contains (at most) one message. Similar properties can also be inferred by inspecting the type associated with objects in our type system.

Parametric Properties. A different approach to checking that programs comply with the protocol of the objects they use is by means of monitoring techniques for parametric properties [29, 32]. Such techniques are based on runtime verification, hence they cannot rule out protocol violations that do not manifest themselves during one particular execution. However, they can be used for verifying rather complex properties involving non-regular protocols (like the fact that a lock is released as many times as it is acquired) and multiple objects (like the fact that a collection does not change while it is being iterated upon). Our type system can use lin-

earity to capture some non-regular protocols. For instance, it requires locks to be released as many times as they are acquired. However, it cannot express contextual properties (like the fact that the lock is released within the same method that has acquired it), nor properties involving multiple objects, since different objects have unrelated types.

9. Concluding Remarks

We have found evidence that the Objective Join Calculus is a natural model for TSOP. The choice of this particular model allowed us (1) to approach TSOP in a challenging setting involving concurrency, object sharing/aliasing, and partial/concurrent state updates; (2) to capture the characterizing facets of TSOP (state-sensitive operations, explicit state change, runtime state querying, object protocols, multidimensional state, aliasing control) with the support of a simple and elegant language of behavioral types equipped with intuitive semantics and subtyping (Section 4); (3) to devise a manageable type system (Section 5) that statically guarantees valuable properties (Section 6) and includes a characterization of safe partial/concurrent state transitions in terms of subtyping (see the side condition of [T-REACTION]).

In this paper we focused on the theoretical foundations of the chemical approach to TSOP and glimpsed at the key aspects that concern its practical realization (Section 7). Below is a non-exhaustive list of extensions and future developments that we find particularly relevant or intriguing.

Aliasing. In our type system, fine-grained aliasing control is realized by the $\otimes$ connective: an object of type (equivalent to) $t \otimes s$ can (actually, must) be used according to both t and s, by possibly parallel processes. Uncontrolled aliasing requires using the exponential $*$. However, neither $\otimes$ nor $*$ express with sufficient precision some forms of aliasing/sharing of objects. It would be interesting to investigate whether and how our type language integrates with other forms of aliasing control [5, 15, 45].

Compiler Optimizations. Efficient compilation techniques for join patterns [31] rely on atomic operations and finite-state automata for tracking the presence messages with a given tag. Our type system paves the way to further optimizations: for example, t_{lock} says that, when the method `release` is invoked, the lock is *for sure* in state `BUSY`. In other words, the reaction involving `BUSY` and `release` can be triggered without requiring an actual synchronization and invocations to `release` compiled as ordinary method calls. The availability of precise information on the behavior of objects can help reducing the amount of locking, simplifying the mechanisms (automata) that detect triggered reactions, and improving the representation of objects with typestate.

Type Inference. Early experiments indicate that it is possible to implement a type inference algorithm for our type system with some minimal help from the programmer. This feature is crucial for the effectiveness of the approach given the use of structural subtyping and the richness of types. We are comforted by the fact that inference of object protocols has been investigated in a number of works (a detailed survey is given in [13]), some of which use specification languages inspired to regular expressions [25] as we do.

Inheritance. Inheritance for concurrent objects is a known source of challenging problems. For the Objective Join Calculus, it has been studied in [19], but with a (non-behavioral) type system focused on privacy. We plan to investigate how our type discipline affects the realization of this feature.

Acknowledgments

The authors are grateful to the anonymous reviewers for their detailed and insightful comments and suggestions and to Philipp Haller for his prompt help in porting Scala Joins to the latest Scala compiler. Inspiring preliminary discussions for this work were hosted at the top of the Roman stairway in Piazza di Spagna. The first author has been supported by the University of Padova under the PRAT projects BECOM and ANCORE. The second author has been supported by ICT COST Action IC1201 BETTY, MIUR PRIN CINA, Ateneo/CSP Project SALT, and RS13MO12 DART.

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of OOPSLA'09*, pages 1015–1022. ACM, 2009.

[2] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *Proceedings of ECOOP'11*, LNCS 6813, pages 2–26. Springer, 2011.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[5] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of OOPSLA'07*, pages 301–320. ACM, 2007.

[6] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[7] P. Calvert and A. Mycroft. Control flow analysis for the join calculus. In *Proceedings of SAS'12*, LNCS 7460, pages 181–197. Springer, 2012.

[8] J. Conway. *Regular Algebra and Finite Machines*. William Clowes & Sons Ltd, 1971.

[9] B. Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.

[10] S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. Technical report, 2015. `https://hal.archives-ouvertes.fr/hal-01155682`.

[11] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in Java-like languages. *Acta Inf.*, 45(7-8):479–536, 2008.

[12] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150. ACM, 2012.

[13] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.*, 22(3):25, 2013.

[14] R. DeLine and M. Fähndrich. Typestates for objects. In *Proceedings of ECOOP'04*, LNCS 3086, pages 465–490. Springer, 2004.

[15] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of PLDI'02*, pages 13–24. ACM, 2002.

[16] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, 1996.

[17] C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Summ. Sch. Appl. Sem.*, LNCS 2395, pages 268–332. Springer, 2000.

[18] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of CONCUR'97*, LNCS 1243, pages 196–212. Springer, 1997.

[19] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *J. Logic Alg. Progr.*, 57(12):23 – 69, 2003.

[20] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, LNCS 2638, pages 129–158. Springer, 2003.

[21] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12, 2014.

[22] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J.Fun. Progr.*, 20(1):19–50, 2010.

[23] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of POPL'10*, pages 299–312. ACM, 2010.

[24] P. Haller and T. V. Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of COORDINATION'08*, LNCS 5052, pages 135–152. Springer, 2008.

[25] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. of FSE'05*, pages 31–40. ACM, 2005.

[26] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of IJCAI'73*, pages 235–245, 1973.

[27] K. Honda. Types for dyadic interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

[28] G. S. V. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In *Proceedings of ACSAC'03*, LNCS 2823, pages 151–165. Springer, 2003.

[29] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Proceedings of PLDI'11*, pages 415–424. ACM, 2011.

[30] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[31] F. Le Fessant and L. Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3):205–224, 1998.

[32] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Autom. Softw. Eng.*, 17(2):149–180, 2010.

[33] Microsoft Research. Cω, 2004. `http://research.microsoft.com/Comega/`.

[34] E. Najm, A. Nimour, and J. Stefani. Guaranteeing liveness in an object calculus through behavioural typing. In *Proceedings of FORTE'99*, volume 156, pages 203–221. Kluwer, 1999.

[35] P. Nienaltowski. *Practical Framework for Contract-Based Concurrent Object-Oriented Programming*. PhD thesis, ETH Zurich, 2007.

[36] O. Nierstrasz. Regular types for active objects. In *Proceedings of OOPSLA'93*, pages 1–15. ACM, 1993.

[37] M. Patrignani, D. Clarke, and D. Sangiorgi. Ownership types for the join calculus. In *Proceedings of FMOODS/FORTE'11*, LNCS 6722, pages 289–303. Springer, 2011.

[38] F. Puntigam. State inference for dynamically changing interfaces. *Comput. Lang.*, 27(4):163–202, 2001.

[39] F. Puntigam. Strong types for coordinating active objects. *Concurrency and Computation: Practice and Experience*, 13(4):293–326, 2001.

[40] F. Puntigam and C. Peter. Types for active objects with static deadlock prevention. *Fundam. Inform.*, 48(4):315–341, 2001.

[41] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of CONCUR'00*, LNCS 1877, pages 474–488. Springer, 2000.

[42] C. V. Russo. The joins concurrency library. In *Proceedings of PADL'07*, LNCS 4354, pages 260–274. Springer, 2007.

[43] C. V. Russo. Join patterns for visual basic. In *Proceedings of OOPSLA'08*, pages 53–72. ACM, 2008.

[44] D. Sangiorgi and D. Walker. *The Pi-Calculus - A theory of mobile processes*. Cambridge University Press, 2001.

[45] S. Stork, P. Marques, and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceedings of OOPSLA'09*, pages 933–940. ACM, 2009.

[46] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, and J. Aldrich. Æminium: A permission-based concurrent-by-default programming language approach. *ACM Trans. Program. Lang. Syst.*, 36(1):2, 2014.

[47] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[48] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter. First-class state change in Plaid. In *Proceedings of OOPSLA'11*, pages 713–732. ACM, 2011.

[49] A. J. Turon and C. V. Russo. Scalable join patterns. In *Proceedings of OOPSLA'11*, pages 575–594. ACM, 2011.

[50] S. West, S. Nanz, and B. Meyer. Efficient and reasonable object-oriented concurrency. In *Proceedings of PPoPP'15*, pages 273–274, 2015.